Open access • Journal Article • DOI:10.1504/IJWGS.2010.033788

A parallel grid-based implementation for real-time processing of event log data of collaborative applications — Source link

Fatos Xhafa, Claudi Paniagua, Leonard Barolli, Santi Caballé

Institutions: Birkbeck, University of London, IBM, Fukuoka Institute of Technology, Open University of Catalonia

Published on: 01 Jun 2010 - International Journal of Web and Grid Services (Inderscience Publishers)

Topics: Grid computing, Event (computing) and Grid

Publisher version: http://www.inderscience.com/offer.php?id
DOI: 10.1504/IJWGS.2010.033788

A parallel grid-based implementation for real-time processing of event log data of collaborative applications

Fatos Xhafa*

Department of Computer Science and Information Systems,
Birkbeck, University of London,
23-29 Emerald Street, London WC1N 3QS, UK
E-mail: fatos@dcs.bbk.ac.uk
*Corresponding author

Claudi Paniagua

IBM GTS, Virtualization and Grid Computing,
Avd. Diagonal, 571 08029, Barcelona, Spain
E-mail: cpaniagua@es.ibm.com

Leonard Barolli

Faculty of Information Engineering,
Department of Information and Communication Engineering,
Fukuoka Institute of Technology (FIT),
3-30-1 Wajiro-higashi, Higashi-ku, Fukuoka 811-0295, Japan
E-mail: barolli@fit.ac.jp

Santi Caballé

Department of Information Sciences,
Open University of Catalonia,
Av. Tibidabo, 39-43, 08035 Barcelona, Spain
E-mail: scaballe@uoc.edu

Abstract: Collaborative applications usually register user interaction in the form of semi-structured plain text event log data. Extracting and structuring of data is a prerequisite for later key processes such as the analysis of interactions, assessment of group activity, or the provision of awareness and feedback. Yet, in real situations of online collaborative activity, the processing of log data is usually done offline since structuring event log data is, in general, a computationally costly process and the amount of log data tends to be very large. Techniques to speed and scale up the structuring and processing of log data with minimal impact on the performance of the collaborative application are thus desirable to be able to process log data in real time. In this paper, we present a parallel grid-based implementation for processing in real time the event log data generated in collaborative applications. Our results show the feasibility of using grid middleware to speed and scale up the process

of structuring and processing semi-structured event log data. The Grid prototype follows the Master–Worker (MW) paradigm. It is implemented using the Globus Toolkit (GT) and is tested on the Planetlab platform.

Keywords: computational grids; grid services; real-time applications.

Reference to this paper should be made as follows: Xhafa, F., Paniagua, C., Barolli, L. and Caballé, S. (xxxx) 'A parallel grid-based implementation for real-time processing of event log data in collaborative applications', *Int. J. Web and Grid Services*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Fatos Xhafa is currently a Visiting Professor at the Department of Computer Science and Information Systems, Birkbeck, University of London. He is an Associate Professor (with tenure) at the Technical University of Catalonia, Spain. His research interests include parallel and distributed algorithms, combinatorial optimisation, distributed programming, Grid and P2P computing. He has widely published in international journals, books and conference proceedings of the research area. He serves the EB of nine peer-reviewed international journals and has also guest co-edited in several international journals. He has served and is currently serving as PC Co-Chair/General Co-Chair of several international conferences and workshops.

Claudi Paniagua is an Associate IT Architect working for IBM Global Services in the organisation of Business Consulting Services Application in Barcelona (Spain). His research interest focus on IT architectures, service-oriented architectures and grid computing.

Leonard Barolli is a Professor in the Department of Information and Communication Engineering, Fukuoka Institute of Technology (FIT), Japan. His research interests include high-speed networks, mobile communication systems, ad-hoc networking, sensor networks, P2P systems, intelligent algorithms (fuzzy theory, genetic algorithms, neural networks), network protocols, agent-based systems, grid and internet computing. He has published widely in refereed journals and international conferences proceedings. He has served and is currently serving as General Co-Chair/PC Co-Chair of many conferences and workshops and as Guest Editor for many journals. He is a member of IPSJ, IEEE and SOFT.

Santi Caballé has a PhD in Computer Science from the Open University of Catalonia (OUC). He is an Associate Professor at the Department of Computer Science, Multimedia and Telecommunication at the OUC. His research focuses on e-learning, software engineering, network technologies and grid technologies.

1 Introduction

In Computer-Supported Collaborative Learning (CSCL) environments, the analysis of the information related to the collaborative group activity is crucial for understanding collaboration and group processes (Dillenbourg, 1999). This information is usually maintained in the form of event log data and is generated automatically by the collaborative application by registering the information related to different types of

actions done by the users of the applications. One such example is the Basic Support for Collaborative Work system (Bently et al., 1997), a collaborative application that generates event log data regarding the connection information as well as actions performed by the users along time activity. The information generated by the collaborative applications can be of a great variety of type and formats (Caballé et al., 2004). Moreover, collaborative applications are characterised by a high degree of user–user and user–system interaction and hence generate a huge amount of information of event log data.

As a matter of fact, the computational cost is the main obstacle to processing the data in real time (Rouillard, 2004), and in real situations, this processing tends to be done offline to avoid harming the performance of the logging application, but as it takes place after the completion of the collaborative activity has less impact on it (Xhafa et al., 2004). Most of the existing approaches in the literature consider a sequential approach for the processing of log data and try to overcome the performance problem by:

i processing for specific purpose (i.e., limiting the quantity of information needed for that purpose)

ii processing of small data samples, usually for research and testing purposes.

Grid technology is increasingly being used to reduce the overall, censored time in processing data by offloading these computationally costly tasks from the computing elements running them onto the Grid. The concept of a computational Grid (Foster and Kesselman, 1999) has emerged as a way of capturing the vision of a network computing system that provides broad access to massive computational resources. Thus, in this paper, we show how to offload onto the grid the online processing of log data from the collaborative application and how a simple MW scheme sufficed to achieve considerable speed-up. The MW scheme is implemented using Grid services of GT. Grid services (Comito et al., 2005; Sun et al., 2007; Huang et al., 2007) are increasingly being used in the development of Grid-enabled online learning applications (Brut and Buraga, 2008). To show the feasibility of our approach, we use the event log data from BSCW system in our real context of Open University of Catalonia[1] though our approach is generic and can be applied for structuring event log data of collaborative applications in general.

The rest of the paper is organised as follows. We give in Section 2 a description of the problem of structuring and processing event log data. The sequential approach for processing of log data is given in Section 3. We give in Section 4 a parallel approach using the MW paradigm and its Grid-services-based implementation in Section 5. Some computational results are given in Section 6 and their evaluation in Section 7. The paper ends up in Section 8 with some conclusions.

2 The problem of structuring and processing event log data

The problem of structuring event log data of collaborative applications in real time can be defined as follows: give structure to the semi-structured textual event log data that an application logs as soon as it logs it and persist the resulting data structure for the later processing by analysis tools.

A special case of this problem, known as log data normalisation or unification that consists in transforming proprietarily formatted log data to a standard log data format, is recently gaining attention from the autonomic computing community (Salfner et al., 2004) as a way to give standard and homogeneous structure to the strongly heterogeneous data that the disparate elements of an IT infrastructure log while they operate.

In fact in the discussion that follows we will be using the terminology and architecture of the Generic Log Adapter (GLA) (Grabarnik et al., 2004), a framework that addresses the problem of real-time log data normalisation and structuring in the IBM's autonomic computing toolkit[2] for building autonomic systems and that has been open source as part of the Eclipse Hyades project. [3]

The GLA is architectured around four components, mapping to each one of the four phases involved in the structuring of log data in real time. These components are briefly explained here:

i *Sensor*: This component monitors one source of plain text log data (i.e., a log file) reading it line by line as it changes as a result of new data being logged. When the sensor has read a preconfigured number of new lines, it passes them to the Extractor component.

ii *Extractor*: The Extractor component receives log data from the sensor and parses it to delimit the messages or event boundaries contained in the log data.

iii *Parser*: The parser component parses the messages/events that are received from the Extractor component and maps them to the target data structure thus giving structure to the log data.

iv *Outputter*: The Outputter component receives the data structure created by the Parser component and persists it for the later processing, e.g., by statistical and mining tools.

More formally, the input of the problem of structuring log data is text, thus it can be modelled using formal language terminology (Hopcroft et al., 2001).

Let the input be represented by a word, ω, from a given alphabet, Σ. The Sensor component reads this word as it is being generated, thus outputs a sequence of subwords of ω, say, $\omega_1, \omega_2, \ldots, \omega_m$. The Extractor component acts on each one of these subwords one at a time, outputting a subword, $E(\omega_i)$, of ω_i, which verifies one simple but important property: it is an independent unit of structure. That is, it contains all the information the Parser component needs to know to be able to transform it into a data structure. Hence, the Parser component acts on it outputting a data structure, $P(E(\omega_i))$, to the Outputter Component who persists it.

It is worth noticing here that the time complexity of the computation of $E(\omega_i)$ is linear. Indeed, this computation is a word recognition problem. Thus, the question of its time complexity is reduced to what kind of languages might an Extractor Component ever had to recognise. It can be argued, due to the nature of log data, that these can only be Regular languages or, in the case of log data with multiple formats, a union of Regular Languages, which is also a Regular Language. Hence, time complexity is linear at most (Baeza-Yates and Gonnet, 1996).

3 The sequential approach for processing of log data

To deal with the problem of extracting useful information from the event logs generated by the BSCW system in real online learning group activities conducted at the Open University of Catalonia, we have developed a simple application in Java, called `EventExtractor`. This application runs offline on the same machine as the BSCW server and uses the daily log files generated by the BSCW server as input so as to:

i identify the event boundaries inside the log file (extractor component)

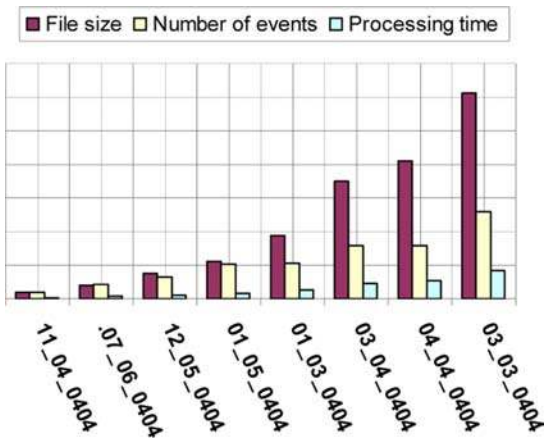
ii map specific information contained in these events about users, objects, sessions, etc., to typed data structures (parser component)

iii store these data structures in a persistent support (outputter component).

Note that as the processing is done offline, there is no need for a sensor component. To analyse the performance of this sequential application and compare it with its parallel Grid-based (see Section 5), we designed a specific test battery in which we used both large amounts of event information and well-stratified short samples consisting of all the existing daily log files making up the whole group activity generated during an academic term of the computer science subject "Software Development Techniques" at the Open University of Catalonia. This course involved two classrooms, with a total of 140 students arranged in groups of 5 students and 2 tutors. On the other hand, other tests involved a few log files with selected file size and event complexity forming a sample of each representative stratum. This allowed us to obtain reliable statistical results using an input data size easy to use.

All our test battery was processed by the `EventExtractor` application executed on single-processor machines involving usual configurations. The battery test was executed several times with different work load to have more reliable results in statistical terms involving file size, number of events processed and execution time along with other basic statistics. The experimental results from the sequential processing of eight event log files are summarised in Figure 1, where for each event log file we show the relative comparison scale for the file size, number of events and the processing time.

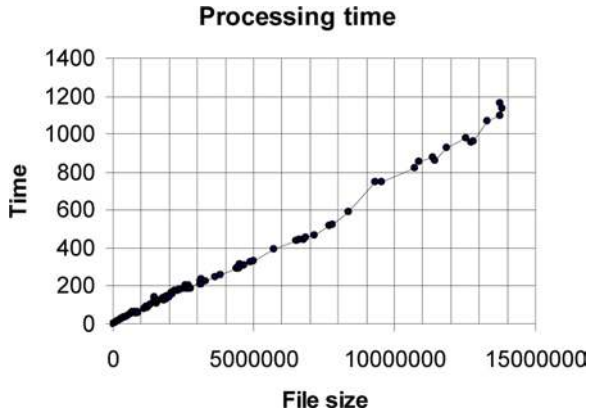
Figure 1 Sequential processing of event log files (file size in bytes, time in seconds) (see online version for colours)



In Figure 1, a certain degree of linearity can be noticed between the number of events and the file size with regard to the processing time.

In a similar way, Figure 2 presents the processing results of over 100 event log files involving file size and processing time showing that the processing time is linear on the size of the log file processed.

Figure 2 Sequential processing time (in seconds) vs. event log file size (in bytes)



This allows us to talk about the processing rate, P (i.e., in Kb/s), of the `EventExtractor` application, and to express the running time (i.e., in seconds) of the `EventExtractor` application with the following formula:

$$T_S(n) = n/P \tag{1}$$

where n is the size (in Kbs) of the event log file.

4 A parallel approach using the master–worker paradigm

In this section, we show how the problem of structuring the event log data can be parallelised using the MW paradigm (Goux et al., 2000; Heymann et al., 2000).

The MW paradigm has been widely used for developing parallel applications. In this model, there are two different types of entities: master and worker. The master decomposes the main task into subtasks (sometimes this reduces to splitting the problem's input into parts) and sends these to the workers. The workers process the subtasks as soon as they receive them and send back the result to the master, which uses them in its main flow of computation.

The MW model has proved to be efficient in developing parallel applications with different degrees of granularity and is particularly useful when the partitioning of the problem is easy to compute and the dependencies between tasks are low. Indeed, this is the case of the problem of structuring plain text log data since:

i the Extractor Component outputs independent units of structure (i.e., messages/events), which means that, if the problem is partitioned using the boundaries of these units, no dependencies between tasks will exist

ii the problem's input can be easily partitioned in these units of structure since, as we have seen, this can be done using regular expressions.

Given all the above, the problem of structuring log data in real time can be naturally parallelised using the MW paradigm by grouping the Sensor and Extractor components at the Master side and leaving the Parser and Outputter components at the Workers side.

One drawback of this approach, however, might be that the Master is not in full control of the size of the task that it sends to workers since events/messages can have arbitrary size. This fact can somewhat reduce the capacity of the Master to play with the task size to better adapt to the different computational capacities of the workers or to their variable workloads, especially if the sizes of events/messages are too large. However, the latter is not a property we could normally expect of log data.

5 Grid services-based implementation

To experimentally test the feasibility of the MW paradigm for parallelising the structuring of event log data, we have implemented a Grid prototype that parallelises the `EventExtractor` application (see Section 3). We used the GT 3.2 and we deployed the prototype on the Planetlab platform. Both GT 3.2 and Planetlab are briefly described next.

The Globus Toolkit [4](GT) is the actual defacto Grid middleware standard. Version 3 of GT (GT3) is a refactoring of version 2 in which every functionality is exposed to the world via a Grid service. Grid services are basically stateful web services. The core of the GT is a Grid service container implemented in Java that leverages and extends the Apache's AXIS web services engine.

Planetlab[5] is an open platform for developing, deploying and accessing planetary scale services. It is, at the time of this writing, composed of 1069 nodes hosted in 494 different sites. Each Planetlab node is an Intel IA32 machine that must comply with minimum hardware requirements (i.e., 1 GHz PIII + 1 Gb RAM) running the same base software, basically a modified Linux operating system offering services to create virtual isolated partitions in the node, called slivers, which look to users as the real machine. Planetlab allows every user to dynamically create up to one sliver in every node, the set of slivers assigned to a user form what is called a slice. It is said that a Planetlab node can run up to 100 concurrent slivers. To test our Grid prototype, we turned Planetlab into a Grid by installing the GT3's Grid service container in every sliver of our slice. Moreover, we implemented the worker as a simple Grid service playing the role of the parser and outputter components and deployed it on the GT3's container of every sliver of our slice. On the other hand, we wrote a simple Java client playing the role of the master and mapping to the sensor and extractor components, which dispatches, using a simple list scheduling strategy, the tasks to the workers by calling the operations exposed by the worker Grid services.

Notice that our objective was not to create a full-blown GT3 MW implementation but rather to show the feasibility of a parallel Grid-based implementation using the MW paradigm for our problem domain as follows.

5.1 Worker's implementation

The worker Grid service publishes an interface with only one operation, namely `processEvents`. The master calls this operation to dispatch a task to the worker. The worker can only do one of these operations at a time (no multithreading). The operation has only one argument: a string containing the textual representation of the events to be processed by that task. The operation returns a data structure containing performance information about the task executed (elapsed time in ms, number of events processed and number of bytes processed). The `processEvents` operation is implemented by wrapping the Java code of the `EventExtractor` application's routine that parses the BSCW log events. In other words, the workers execute exactly the same java bytecode to process the log events as the `EventExtractor` application. This makes possible the performance comparison between the sequential and Grid approaches.

5.2 Master's implementation

The master is essentially a 'normal' Java application that reads from a configuration file:

1 the folder that contains the event log files to process

2 the available workers

3 the number of workers to use

4 the size of the task to be dispatched to each worker expressed in number of events.

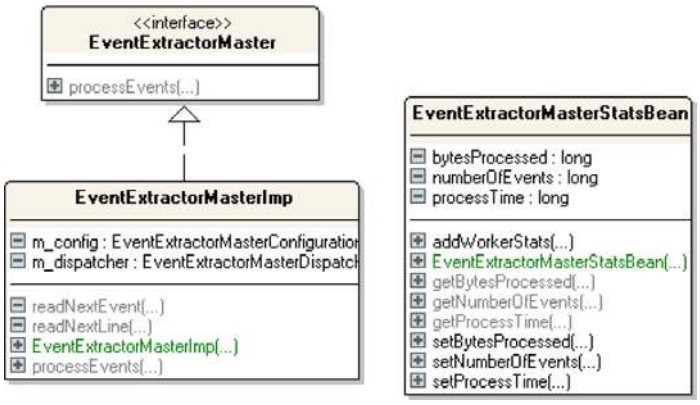
The master then proceeds as follows: peeks as much workers as needed from the configuration file and puts them all in a queue of idle workers, then enters a loop reading line by line (i.e., sensor component) the data contained in the event log files located in the folder specified in the configuration file, and parsing each one of these lines in search of the boundaries between events to extract those (i.e., extractor component). Every time the master reads a number of events equal to the size of the task specified, it creates a thread that gets a worker from the queue of idle workers (synchronously waiting for a worker if the queue is empty) and synchronously calls the worker's `processEvent` operation. Once the call to the worker returns, the worker is put back into the queue of idle workers. The master exits the loop when all events in the event log files have been read and all the tasks that were dispatched are completed.

The Master implements the `EventExtractorMaster` interface, which has a single operation to process events:

```
Final public EventExtractorMasterStatsBean processEvents() throws Throwable
```

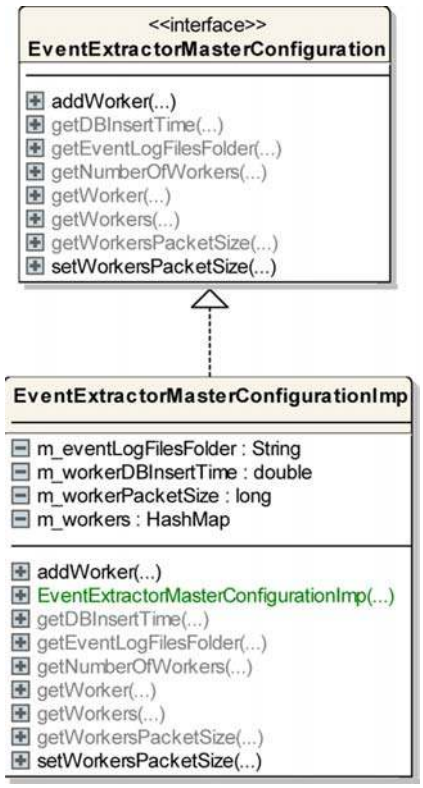
The operation returns an `EventExtractorMasterStatsBean` instance containing some performance statistics about the execution of the operation (see Figure 3).

Figure 3 Master's interface (see online version for colours)



The `EventExtractorMasterImp` class implements the `EventExtractorMaster` by aggregating an `EventExtractorMasterConfiguration` instance that configures its operation and an `EventExtractorMasterDispatcher` instance to dispatch tasks to workers. An `EventExtractorMasterConfiguration` just exposes operations to set and retrieve configuration parameters of the behaviour of a master (see Master's configuration in Figure 4).

Figure 4 Master's configuration (see online version for colours)



On the other hand, the EventExtractorMasterDispatcher interface defines operations to dispatch tasks to workers and synchronise with them.

```
 public void dispatchEventsToWorker(String
events, long nEvents,
       double workerDBInsertTime, EventExtractorMasterStatsBean stats)
       throws Throwable;

public void waitForAllDispatchsToFinish() throws Throwable;
```

The dispatchEventsToWorker operation synchronously sends a sequence of events to an available worker while waitForAllDispatchsToFinish operation does not return until all pending dispatchEventsToWorker operations have returned.

There is a base implementation of this interface from which two final implementation classes extend. The base implementation class uses an instance of an Event ExtractorMasterIdleWorkerQueue interface to implement the queue of workers. Each one of the classes that extend from the base implementation class specialises the behaviour by aggregating a different implementation of the interface EventExtractorMasterIdleWorkerQueue at construction time.

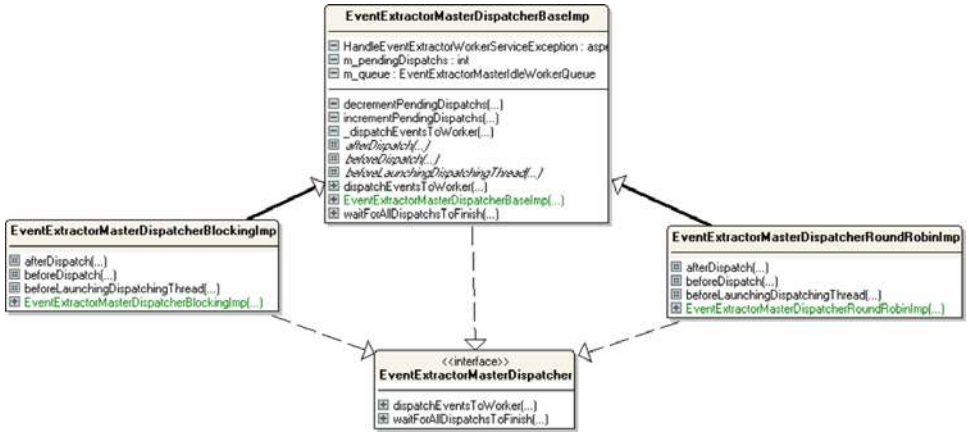
The EventExtractorMasterDispatcherBlockingImp implements the operation dispatchEventsToWorker by spawning a thread as follows (error treatment omitted for simplicity):

```
final public void dispatchEventsToWorker(final String
    events, final long nEvents, final double
    workerDBInsertTime,
    final EventExtractorMasterStatsBean stats) throws Throwable
    { this.incrementPendingDispatchs();
    this.beforeLaunchingDispatchingThread();
    Thread thread = new Thread(new Runnable() {
    public void run() {
        EventExtractorMasterDispatcherBaseIm
        p.this
        ._dispatchEventsToWorker(events, nEvents, workerDBInsertTime, stats);
        }
    });
thread.start(); }

private void _dispatchEventsToWorker(String events, long nEvents,

    double workerDBInsertTime, EventExtractorMasterStatsBean masterStats)
    throws Exception {

    EventExtractorWorker worker = null;
    worker = m_queue.getNextWorker();
    this.beforeDispatch(worker);
    EventExtractorWorkerStatsBean
    workerStats =
    worker.processEvents(events.toString(), workerDBInsertTime);
    this.afterDispatch(worker);
    this.decrementPendingDispatchs();
}
```

5.3 Master's dispatching strategies

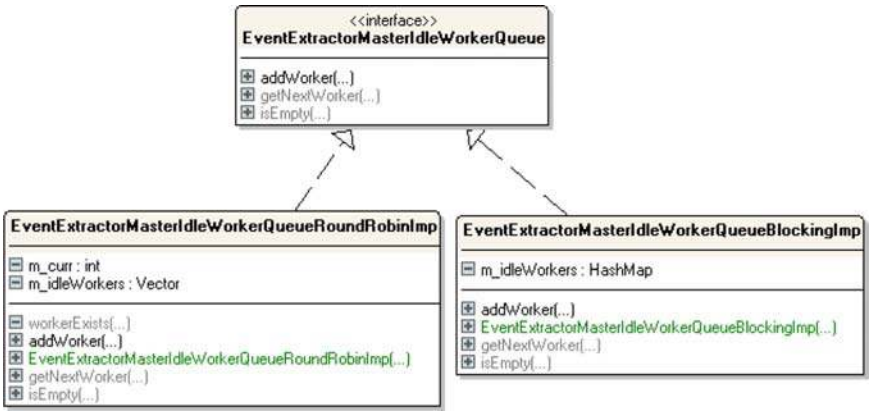
We have implemented two different dispatch strategies that specialise the base dispatch implementation by overriding the method afterDispatch and by instantiating a different implementation class of EventExtractorMasterIdleWorkerQueue interface at construction time (see Figure 5).

Figure 5 Master's dispatching strategies (see online version for colours)



The `EventExtractorMasterDispatcherBlockingImp` dispatch strategy uses the `EventExtractorMasterIdleWorkerQueueBlockingImp` class to implement its queue of idle workers (see Figure 6). This class implements the getNextWorker operation by blocking until the queue of idle workers is not empty, it then picks up a worker removing it from the queue and returns it to the caller. Observe that the instance of `EventExtractorMasterDispatcherBlockingImp` then calls the processEvent operation on this worker and after that puts back the worker in the queue.

Figure 6 Master's implementations queues



The `EventExtractorMasterDispatcherRoundRobinImp` dispatch strategy (see Figure 6) uses the `EventExtractorMasterIdleWorkerQueueRound RobinImp` class to implement its queue of idle workers. This class maintains a circular

counter that points to the 'next available' worker and implements the getNextWorker operation just by incrementing the counter modulo the size of the queue and returning the worker pointed by it. Notice that this dispatching strategy floods the workers with tasks without waiting for them to become idle in a round-robin scheme.

Notice that the scheduling strategy (i.e., list scheduling) favours the faster nodes and thus it is appropriate for an environment where worker machines have unpredictable workloads as the Grid, however, in a more homogeneous workload environment a simple static round-robin scheduling strategy could be more efficient.

6 Computational results

In this section, we present the experimental results of our Grid prototype. To evaluate them, it is important to understand how they were collected and what was measured. Basically, we measured parallel speed-up and efficiency for different executions of the parallel processing of 1000 events using different number of workers, p, (physical scaling) ranging in 2, 4, 8, 16 and different task sizes, n_s (i.e., in number of events) ranging from 1 event to $\lceil 1000/p \rceil$ events.

Parallel speed-up is used to measure the performance gain from a parallelised execution of the application over its serial execution, defined as follows:

$$S(n, p, s) = T_S(n)/T_P(n, p, s) \tag{2}$$

where n is the size of the input, s is the task size, $T_s(n)$ is the total running time of the sequential execution for an input of size n and $T_P(n, p, s)$ is the total running time of the parallel execution for an input of size n, using p workers with a task of size s.

Parallel efficiency measures the degree of utilisation of the computing resources involved in the parallel computation and is defined as the speed-up divided by the number of computing resources (i.e., workers):

$$E(n, p, s) = S(n, p, s)/p. \tag{3}$$

To characterise the speed-up of our prototype, we run k different executions for each combination of number of workers and task sizes that we tested and then applied the following formula:

$$S_o(n, p, n_s) = \frac{\sum_{i=1}^{k} \sum_{j=1}^{p} T_{pE_{ij}}(n)}{\sum_{i=1}^{k} T_{MWi}(n, p, n_s)}, \tag{4}$$

where $T_{pEij}(n)$ is the time spent by the jth worker executing its `processEvent` operation in the ith execution, while $T_{MWi}(n, p, n_s)$ is the total running time of the master in the ith execution.

Note that we use averaged values to compute the speed-up and that the total sequential execution time was not computed by running the `EventExtractor` application in an arbitrary machine, but by summing up the times spent by the workers executing its `processEvent` operation. This time can be thought of as the serial execution time of the application on a hypothetical machine with varying computational power and workload equivalent to the ones experimented by the workers during the parallel execution. Thus, we can have a more realistic idea of the speed-up achieved.

We show in Figure 7 and Figures 8–11 the main results of our experiments. Figure 7 shows how the observed speed-up and parallel efficiency of our prototype scaled with the number of workers for a fixed task size of 25 events. For the rest of figures, for each number of workers we tested, it is shown how the observed speed-up varied with the size of the task.

Figure 7 Speed-up and efficiency vs. No. of workers for a task size of 25 events

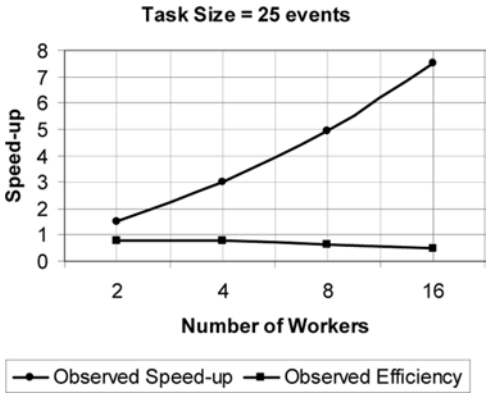


Figure 8 Speed-up vs. task size for 2 workers

Figure 9 Speed-up vs. task size for 4 workers

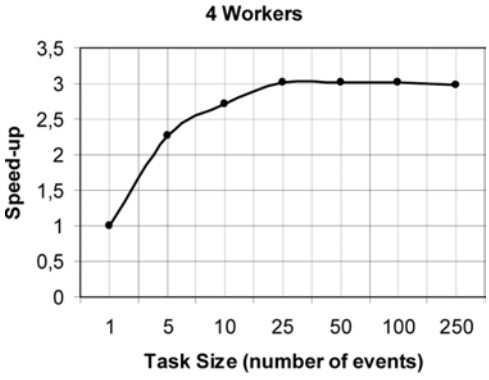
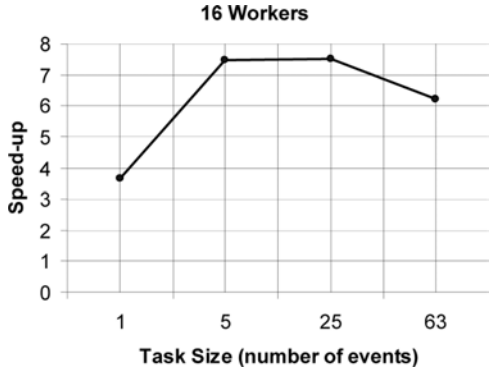


Figure 10 Speed-up vs. task size for 8 workers

Figure 11 Speed-up vs. task size for 16 workers



7 Analysis of the results

From the results obtained, it can be concluded that a reasonable speed-up has been achieved in every tested configuration. However, we observe that the parallel efficiency decreases with the number of workers (see Figure 7). This could be explained due to the fixed size of the input to 1000 events since the speed-up seems to grow with the task size except for values near $\lceil 1000/p \rceil$ where it begins to decrease.

As can be seen from the figures, the threshold in the task size reduces from 25 for 4 and 8 workers (see Figures 9 and 10) to just 5 (see Figure 11) in case of 16 workers. Indeed, for too small values of the task size, the overhead introduced by the transmission protocol when sending the parts to the workers is noticeable and the implemented list scheduling strategy may be spending too much time waiting for completion notifications. On the other side, values of the task size close to $\lceil 1000/p \rceil$ considerably diminish the attainable degree of concurrency; however, it is here where increasing the size of the problem, n, could be useful.

Our results show the feasibility of parallelising the problem of structuring any plain text event log data, achieving considerable speed-up, provided that

1 the structuring algorithm's running time function, $f(n)$, be of strictly lower order than the transmission time function, n/B, that measures the time required to transmit a piece of data of size n for a bandwidth B (i.e., $f(n) = \omega(n/B)$)

2 the log data can be easily parsed (i.e., with regular expressions at most) to be broken in independent units of structure (i.e., message/events).

Log data, as well as structuring algorithms, especially the ones that can be found in generic log data structuring/normalising frameworks (such as the GLA, which are implemented using regular expressions) satisfy these assumptions most of the time.

We finally note that although the results of the experimental study are dependent on the form of the BSCW event log files, the parallelisation strategy presented in this paper is generic and can be applied to parallelise the structuring of collaborative application's events log data.

8 Conclusion and future work

In this paper, we first have motivated the need to process in real time the large amount of information generated in collaborative applications for important purposes such as awareness, feedback, assessment, workspace design and interaction analysis. Then, we have shown how to use a grid-based approach to overcome the drawbacks of existing approaches. To this end, we have presented a proof of concept grid implementation to speed up the processing of the event log data generated in collaborative applications. We have particularised the approach for the case of event log data of the BSCW system.

The results show the feasibility of parallelising the problem of structuring any plain text event log data, achieving considerable speed-up. On the other hand, we want to emphasise that although the parallelisation of event log data processing could have been done with any other distributed Java-based technology, doing it with Globus Toolkit 3 offers several advantages:

1 opens the door to a very costly effective and powerful way of harnessing computer power as any machine capable of running the Java platform can be easily turned into a worker by just installing the GT3 Container on it and deploying our worker service

2 it is also very easy to achieve a simple but working and performing solution that can be incrementally extended to a full-blown grid solution that may reuse the many powerful features of the GT such as fast data transfer, notification and dynamic discovery of workers.

In fact, there are many aspects of the prototype that we plan to enhance in the near future, among them, fault-tolerance, dynamic discovery of workers and the possibility of implementing the communication between the master and the workers by other means than the default transport mechanism (i.e., SOAP over HTTP) used by GT3. In particular, we would like to use gridFTP to explore the possibility of sending large size tasks to the workers and using OGSI notification to communicate asynchronously to the master the completion of tasks by the workers, which would result in far more scalable way of keeping track of task completion than the current approach of having a

thread waiting for each pending task. To achieve full scalable implementation, the synchronisation should be implemented by leveraging the Globus publish/subscribe event infrastructure.

Finally, we plan to provide our parallel application with better scheduling strategies that would result in improvement of parallel speed-up. All in all, the promising experimental results obtained together with the powerful features provided by the GT encourage us to keep working on to extend the current prototype to a full-blown Grid implementation capable of speeding and scaling up the real-time processing of collaborative group activity log data by harnessing resources in the dynamic, opportunistic and heterogeneous distributed environment such as computational grids.

Acknowledgement

Fatos Xhafa's research work is supported by a grant from the General Secretariat of Universities of the Ministry of Education, Spain.

References

Baeza-Yates, R.A. and Gonnet, G.H. (1996) 'Fast text searching for regular expressions or automaton searching on tries', *Journal of the ACM*, Vol. 43, No. 6, pp.915–936.

Bentley, R., Appelt, W., Busbach. U., Hinrichs, E., Kerr, D., Sikkel, S., Trevor, J. and Woetzel, G. (1997) 'Basic Support for Cooperative Work on the World Wide Web'. *Int. J. Human-Computer Studies*, Vol. 46, No. 6, pp.827–846.

Brut, M. and Buraga, S. (2008) 'An ontology-based approach for modelling grid services in the context of e-learning', *Int. J. Web and Grid Services*, Vol. 4, No. 4, pp.379–394.

Caballé, S., Xhafa, F., Daradoumis, Th. and Marqués, J.M. (2004) 'Towards a generic platform for developing CSCL applications', *Int. Workshop on Collaborative Learning Applications of Grid Technology (CLAG'2004)*, Part of the *4th IEEE/ACM Int. Symp. on Cluster Computing and the Grid (CCGrid'2004)*, USA, IEEE, 2004 (Proceedings CD, 0-7803-8430-X/04/IEEE).

Comito, C., Talia, D. and Trunfio, P. (2005) 'Grid services: principles, implementations and use', *Int. J. Web and Grid Services*, Vol. 1, No. 1, pp.48–68.

Dillenbourg, P. (1999) 'What do you mean by collaborative learning?', in Dillenbourg, P. (Ed.): *Collaborative-Learning: Cognitive and Computational Approaches*, Elsevier, Oxford, pp.1–19.

Foster, I. and Kesselman, C. (Eds.) (1999) *The Grid 2e, 2nd Edition Blueprint for a New Computing Infrastructure*, Morgan-Kaufman, Chapter 2, pp.13–24.

Goux, J., Kulkarni, S., Yoder, M. and Linderoth, J. (2000) 'An enabling frame-work for master-worker applications on the computational grid', *Proceedings of the 9th IEEE international Symposium on High Performance Distributed Computing (August 01 – 04, 2000), High Performance Distributed Computing*, IEEE Computer Society, Washington, DC, p.43.

Grabarnik, G., Salahshour, A., Subramanian, B. and Ma, Sh. (2004) 'Generic adapter logging toolkit', *First International Conference on Autonomic Computing (ICAC'04)*, pp.308–309.

Heymann, E., Senar, M., Luque, E. and Livny, M. (2000) 'Adaptive scheduling for master-worker applications on the computational grid', *Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, LNCS, Vol. 1971, pp.214–227.

Hopcroft, J.E., Motwani, R. and Ullman, J.D. (2006) *Introduction to Automata Theory, Languages, and Computation*, 3rd ed., Addison-Wesley Longman Publishing Co., Inc.

Huang, W., Wu, Y., Yuan, Y., Liu, J., Yang, G. and Zheng, W. (2007) 'Parallel programming over ChinaGrid', *Int. J. Web and Grid Services*, Vol. 3, No. 4, pp.480–497.

Rouillard, J.P. (2004) 'Real-time log file analysis using the simple event correlator', *Proceedings of the 18th USENIX Conference on System Administration (Atlanta, GA, November 14 –19, 2004), System Administration Conference*, USENIX Association, Berkeley, CA, pp.133–150.

Salfner, F., Tschirpke, S. and Malek, M. (2004) 'Comprehensive logfiles for autonomic system', *18th International Parallel and Distributed Processing Symposium (IPDPS'04) – Workshop 11*, Vol. 12, p.211b.

Sun, H., Huai, J., Hu, Ch. and Li, Q. (2007) 'Design and implementation of an enhanced grid service container in the CROWN grid', *Int. J. Web and Grid Services*, Vol. 3, No. 4, pp.403–423.

Xhafa, F., Caballé, S., Daradoumis, Th. and Zhou, N. (2004) 'A grid-based approach for processing group activitylog files', *Proc. of the GADA'04*, Cyprus, LNCS Vol. 3292, pp.175–186.

Notes

[1]http://www.uoc.edu

[2]http://www-106.ibm.com/developerworks/autonomic/overview.html

[3]http://eclipse.org/hyades/

[4]http://www.globus.org

[5]http://www.planet-lab.org